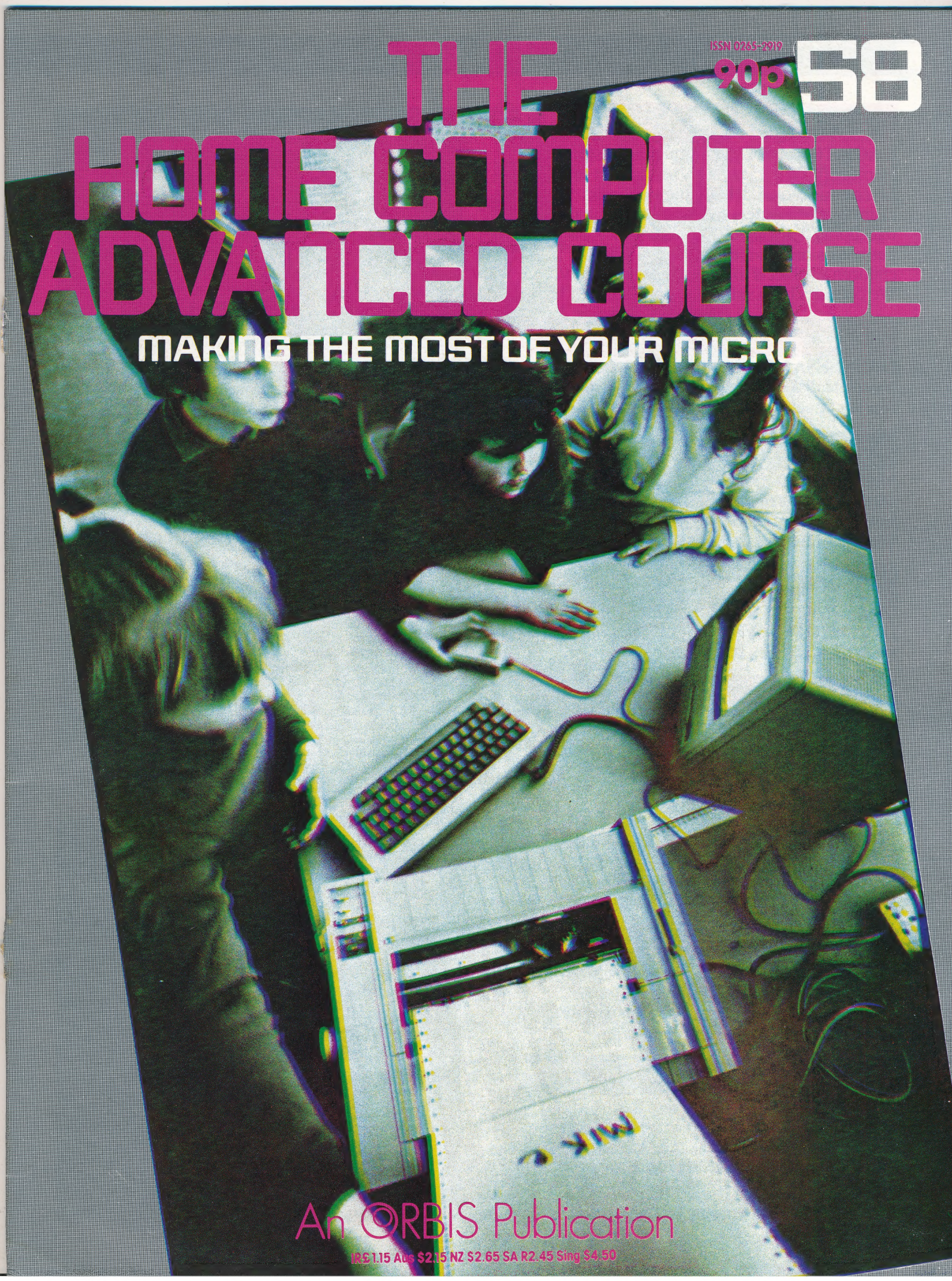
ISSN 0265-2919

90p

58

# THE HOME COMPUTER ADVANCED COURSE

## MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

# CONTENTS



## Next Week

- **The Penman Plotter can be used as a digital plotter, a turtle or a mouse. We take a look at this versatile machine.**
- **We show how the remaining minor contingencies of the voyage are coded in our New World simulation game.**
- **Having now completed the construction of the robot arm we begin to design software to accompany it.**


## QUIZ

1) Is it preferable for an electronic device to have a high or low 'noise margin'?
2) What is the purpose of 'packing' an array in PASCAL?
3) Is it possible to run dBase II on an Apple II computer?
4) How does the Commodore 64's serial port differ from the RS232 standard?

**Answers To Last Week's Quiz**

**1)** 'Keys' refer only to a tiny area of the database, and so when a key is used for searching, the computer is able to narrow its search down to a small number of records, rather than searching through every record.

**2)** The Wafadrive operating system monitors the error-trapping routine of the Spectrum and pages itself in when a WOS command occurs.

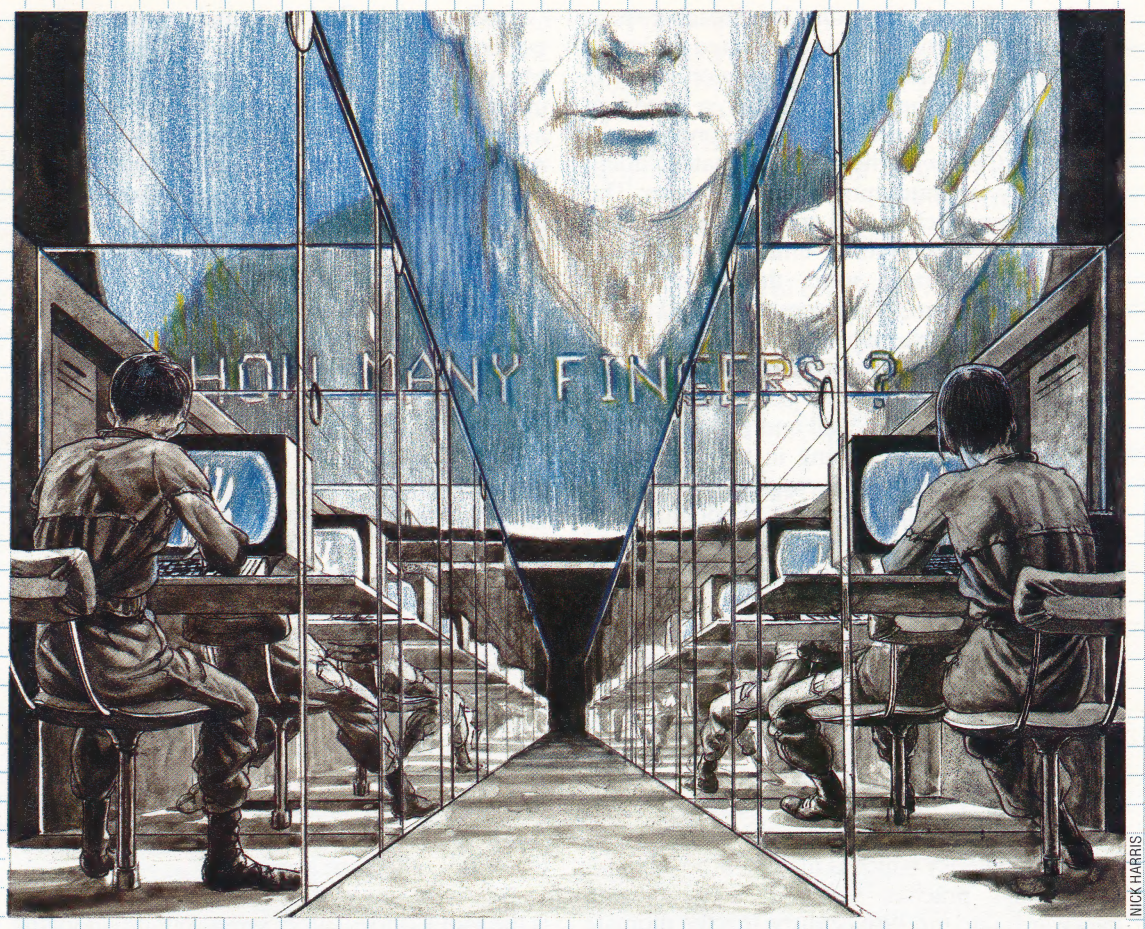
**3)** The unstructured programs written by students using BASIC led Borge Christensen to develop the COMAL language.

**4)** By knowing the contents of the BASIC pointers, we can calculate the start address of BASIC programs and, therefore, alter or call it according to our needs.


**Editor** Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Technical Editor** Steve Colwill; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Steve Malone; **Sub Editor** Jon Kaye; **Contributors** Geoff Bains, Nick Walsh, Dr Antonia Jones, Steve Malone, David Mudd, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

**HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE** – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
**UK/EIRE** – Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
**ADDRESS FOR BINDERS AND BACK ISSUES** – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
**NOTE** – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
**ADDRESS FOR SUBSCRIPTIONS** – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY TONY SLEEP ELECTROPHOTOGRAPHY BY LAURIE-RAE CHAMBERLAIN




NICK HARRIS

**Interactive Indoctrination**
The image of CAL as a forerunner of an Orwellian nightmare owes much to the work of early behavioural psychologists and little to present-day developments. Compact disc technology, interactive video and networks all hold out much promise for the future of computer-assisted learning

# BACK TO BASICS?

**The term 'computer-assisted learning' (CAL), though now applied to almost any learning process accompanied by a computer, actually refers more precisely to the tutorial aspect of teaching. Here, we look at CAL in this light, pointing out the historical and present day opinions, attitudes and practices.**

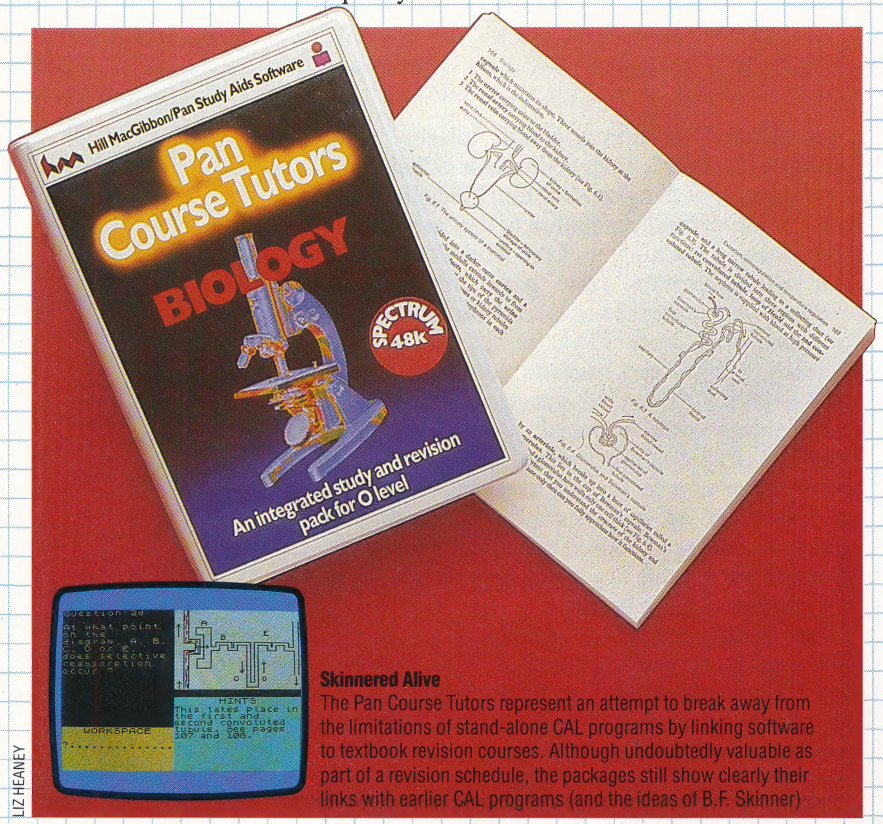
CAL is an acronym for computer-assisted learning. It is also sometimes known as CAI, for computer-aided instruction, or CBL, for computer-based learning. The term has been used in widely differing contexts, denoting anything from a particular type of tutorial software to any general educational activity involving computers. In this article, CAL is used in its narrower sense — namely, to describe tutorial type programs, and *not* games, simulations and children's programming. In CAL, the computer is used to transmit information, test to see whether it has been learnt, supervise drill and practice, and act as an 'intelligent', programmed learning machine.

In Britain, current opinion places CAL programs alongside such redundant educational activities as scratching on slate and filling inkwells. Much of the distaste dates back to the early days of 'programmed learning' and the works of Skinner and his successors (see page 1001). This preconceived reaction has been strengthened by the release of poor CAL programs over the succeeding years and teachers now treat it with deep suspicion. 'The CAL programs we've had in school have been abysmal', comments Alan Coode, recipient of the Sunday Times award for promoting computer literacy in education. 'They are based on an impoverished philosophy of education, and see children in behaviourist terms. I'd need a very strong argument to persuade me to buy another CAL program.'

The history of CAL development is not encouraging. At the beginning of 1980, with less than 100 microcomputers in British schools, there was clearly no market for educational software. Teachers who did have access to computers had to write their own prgrams in BASIC, and since they were not experienced programmers their software

was, in a word, primitive. One head teacher spent days writing a program that generated addition, subtraction, multiplication and division sums. If the child figured the correct answer, a large tick appeared on the screen; if it was wrong, a cross appeared. Children who normally hated 'doing sums' enjoyed it much more on the computer. The program was proclaimed a great success, and no matter how unimaginative, unfriendly and bug-ridden the programs were, the children still enjoyed using them. They were captivated by the medium, not the message. But it didn't occur to anyone that children were using an expensive resource for an activity that could have just as easily been done with pencil and paper.

In the interests of staying in touch with the times, many teachers have accepted CAL programs they would probably condemn under other circumstances. As a result of this misplaced enthusiasm and commercial manipulation, there was a deluge of CAL software in the early 1980s. In 1982, tutorial programs, including drill and practice, formed almost 80 per cent of the *School Micro Directory*, a major educational software catalogue in the United States. A backlash against this trend soon became apparent. This opposition to CAL was summed up by David Chandler in his book *Young Learners and the Microcomputer:* 'The microcomputer', he writes, 'is a tool of awesome potency which is making it possible for educational practice to take a giant step backwards into the 19th century. It is the ultimate weapon of those who want to "get back to basics", allowing them to process children from "input" to "output" in terms of "behavioural objectives" and "quality control" . . .'




**Skinnered Alive**
The Pan Course Tutors represent an attempt to break away from the limitations of stand-alone CAL programs by linking software to textbook revision courses. Although undoubtedly valuable as part of a revision schedule, the packages still show clearly their links with earlier CAL programs (and the ideas of B.F. Skinner)

LIZ HEANEY

Sadly, much CAL material provides strong evidence in support of such an extreme dismissal. Many programs reinforce bad habits, encourage unimaginative answers, and allow for not much more than either right or wrong answers. Although it is claimed that CAL allows children to learn at their own pace, it is more precise to say that they are learning at the computer's pace.

There are signs, however, of a resurgence of interest in computer-assisted learning. Most home computers are used by 11 to 18 year old males for playing arcade games, and this market has helped to establish a large hardware base that is extensively used by people in full-time education. As an extension, a large market has recently appeared for CAL software, to support in-home study for school examinations. One major British educational software house, Hill MacGibbon, has recognised this new market and invested in producing some respectable CAL programs to support the O-Level examination curriculum.

In common with other software companies, Hill MacGibbon has identified two main areas of educational software. The first is the package designed for a specific purpose, to enable the individual to prepare for academic or professional examinations. The second area concerns education in its broadest sense, in which the computer can be used by students to explore topics not otherwise accessible.

A good example of the first category of product is the Pan Course Tutors series, published jointly by Pan Books and Hill MacGibbon. There are six titles in the series — Maths, Biology, Chemistry, Economics, French and Physics — and each is accompanied by a textbook. This avoids the problem of trying to include large blocks of information within the program, leaving the software free to concentrate on the interactive aspect of learning. The book/program format is likely to remain with us as long as home users stick to eight-bit machines, but once processors capable of addressing more memory make their way into the home, textbooks will become largely redundant. The most likely compromise is for windowing software to enable the presentation of information and pupil interaction on the same screen simultaneously.

In the Pan Course Tutors series, the programs ask questions on the chosen subject. Benefits of using the computer include the psychological advantage of not being able to turn to the answer page, and, if a wrong answer is submitted, a hint to the solution is given — perhaps a graphic clue or a prompt like 'Did you remember to include the brackets?'. If a second incorrect answer is given, a further clue is displayed — usually a reference to a page or section in the text book so that the student can check references. Responses are both timed and recorded so that the program can analyse the results and even suggest areas for further study. By switching between the textbook and the computer, students can work in their own time and at their own pace without competitive pressure.

Computer-assisted learning can also be used to teach children how to use the computer as a tool. A program that takes children through the different functions of a word processor, for example, is invaluable. It is easier to follow a tutorial on the monitor — typing in the appropriate commands and getting feedback — than plodding through a manual. The Apple Macintosh uses this type of presentation on disk, coupled with an audio cassette, to teach about the features of the computer, the word processor and graphics programs.

Unfortunately, the growth of the potential market for educational software has already prompted some extravagant claims for poor products, which, of course, inhibit the development of good software. One well-known publisher produced an expensive package of four programs, one of which was a game of rounders. The purpose of the package, according to the notes supplied with it, was that it could be used on a rainy day to teach children the rules of the game. But since the programmer took so many shortcuts in designing the package — including leaving out many of the finer points of the rules — children attempting to learn rounders are likely to have a hard time of it. Other outrageous claims made concerning its educational benefits included: 'The program gives plenty of opportunity for keyboard typing . . . It starts with one of the two players having to enter his or her name.' It also claims to encourage an interest in sports. The game is written in BASIC and is so slow that even the excitement of an 'uneducational' video game is lacking.

As the novelty of computers wears off, teachers and educationalists will look more critically at software and express their needs more clearly. We can probably expect to see fewer CAL programs, but these will be of a higher standard, useful in special areas of the curriculum or for individuals learning at home.

The dividing line between what is defined as computer-assisted learning and other educational software — with which the child has more control over what is happening — does not exist in reality. Tutorial type presentation can be mixed with simulations and experiments that might otherwise be impossible to carry out in the school laboratory. And even the staunchest of critics must admit that they are an obvious aid to students preparing for examinations or studying outside an institution.

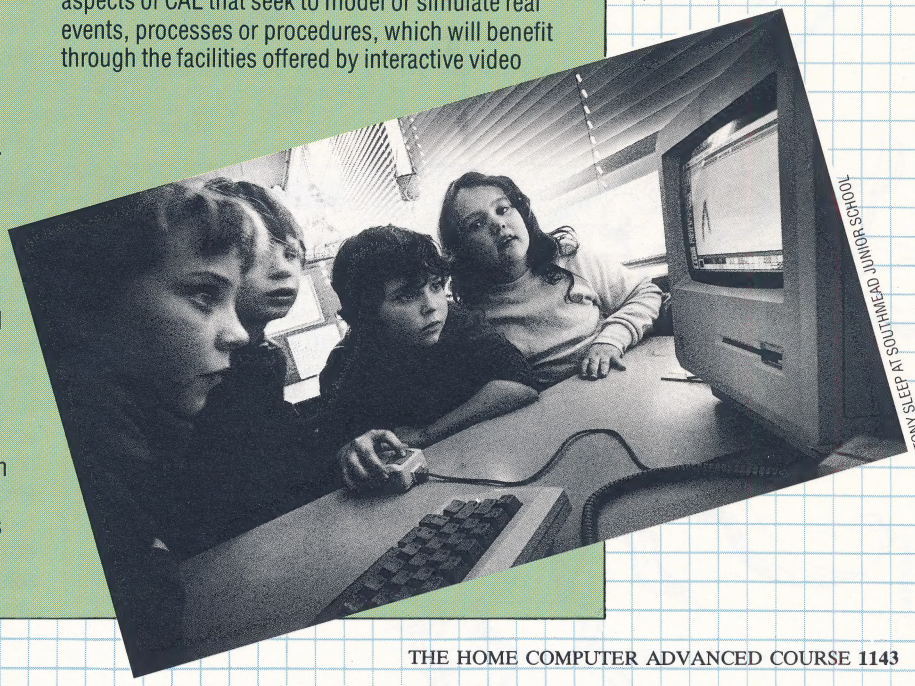
## In Rehearsal

Present CAL software is useful for teaching what standardised tests measure, but there is little in this software to engage the learner's imagination. For the CAL of tomorrow to be an exciting and valuable aid to education, three factors need to change. Antiquated educational techniques, such as drill-learning, on which so many CAL programs are based, need to be discarded. More powerful computers are needed — the present micro-computers with their small memories cannot support the range of software or power necessary for interactive computer-assisted learning. But the most limiting factor has been that the design of the programs has been in the hands of those who understand programming — not necessarily those who understand how people learn.

An early attempt to change this was PILOT, a computer language that was designed to enable teachers to produce learning packages for children. Work in this area has continued at Xerox Palo Alto Research Centre in California. Here a Rehearsal World has been developed using a visual programming environment. It was work at this laboratory that led to the development of the Apple Macintosh, and Rehearsal World appears to extend the concept underlying that machine. Non-programmers can use Rehearsal World to produce educational software in SMALLTALK. The designer moves 'performers' around on a 'stage', teaching them how to interact by sending 'cues', all within an interactive graphic environment. It attempts to use this environment in a way similar to turtle graphics in LOGO. Rehearsal World needs a lot of memory, but it does indicate one direction CAL will be taking in the future.

Even now a computer offers the learner many routes through a learning program and the potential possibilities using interactive video are vast. A combination of an authoring language such as MICROTEX or Apple SUPER PILOT, and interactive video would give the educator unprecedented control over the learning situation.

In the school, interactive video can be used to teach classes, providing a sound and picture source, or by small groups of children using the disc as a tutor. Individualised learning using the full interactive capabilities for open-ended study is certainly a possibility. A resource base for data access could be in the library. It is the creative aspects of CAL that seek to model or simulate real events, processes or procedures, which will benefit through the facilities offered by interactive video



TONY SLEEP AT SOUTHMEAD JUNIOR SCHOOL

**In Performance**
The spread of new 'graphic environment' operating systems like Apple's Macintosh (shown below) and Digital Research's GEM is bringing considerable processing power under the control of formerly computer-illiterate users. The implications of the new 'user-friendly' machines for computer-assisted learning are enormous, since previously all CAL programs had to work on the assumption that the user had no control over the machine other than to answer questions when asked

# MAN OVERBOARD

**During our journey to the New World, we will find that a series of random events can occur. These affect supplies of provisions and trading goods, the number and strength of the crew and, consequently, the length of the voyage. One minor and one major event can happen each week, and in this instalment we begin the programming of the minor events.**

So that we can simulate the difficulties and everyday accidents encountered on long sea voyages in the 16th century we must make allowances within the program for the occurrence of a number of events. Some of these events may assist the sailors, such as a favourable wind blowing, but others can have a deleterious effect on the voyage, such as losing crew members or provisions overboard.

There is a total of 16 possible events, only two of which can occur in a given week. In this instalment we will write a subroutine to select randomly an event from the list of contingencies, including the code for the first five contingencies. In the previous instalment, a piece of code was inserted to set randomly the crew strengths. Remove lines 601 to 604 before typing in Module Six. The new code has several parts: an initialisation section, two lines in the main journey loop, and a subroutine to select a random event and deal with five of those events.

It is important that each random event occurs only once during the journey. To achieve this, an array, RR() is DIMensioned in line 42, containing 16 elements that correspond to the 16 possible events. Each element is set to 0. When an event occurs, the corresponding element will be set to 1, so the program will recognise and not repeat it. Variables that indicate co-ordinates within a program are known as flags. It is necessary to keep a count of the events as they happen to stop the program endlessly searching through the array for another event when they have all occurred. The number of events is recorded in the variable RC, which is incremented each time an event occurs. The number of possible events is held in RM. Note that we will be adding more events to the program in the next instalment, so it will be necessary to change the value of RM in the future, to correspond to the new amount.

Line 860, in the main journey loop, calls to the subroutine at line 5500, which generates a random event. This call is inserted in the main loop so that the event occurs after the Captain's Log has given the estimated length of the remaining voyage. Subroutine 5500 selects which event will occur each week. If all the events have already occurred, the game returns to the main program without generating an event. Line 5502 checks this by comparing RC, the number of events already occurred, to RM, the number of events in the program.

## THE RANDOM EVENTS ROUTINE

A random number between 1 and the number of possible events, RM, is generated at line 5510. Line 5520 checks the array RR() if the element for that event has been set to 1, signifying that it has already occurred. If so, it returns to the main program without generating an event. When a new event has been selected, the corresponding flag in the array RR() is set to 1 by line 5523 to prevent reselection. The same line increments the count of events, RC, by one.

In line 5525, the ON X GOTO statement is used to direct the program to the appropriate code for the selected event. ON . . . GOTO and ON . . . GOSUB are, together, a way of replacing a number of

IAN McKINNELL

IF . . . THEN statements. Each command is followed by a list of line numbers. If the variable is 1, then the first line number is selected from the list. If its value is 2, then the second line number is selected, and so on. Note that the Spectrum does not support ON . . . GOTO and ON . . . GOSUB. Refer to the Basic Flavours box for this Module.

We have five statements corresponding to the five events we are covering in this instalment. The first event, a person being washed overboard, occurs at line 5540. The last four numbers in the ON . . . GOTO list are the same, so if the random number is 2,3,4 or 5, the program is sent to the same place. The four contingencies at elements 2 to 5 deal with provisions washed overboard, and are covered by the same piece of programming starting at line 5570.

If the first line number is selected, a message that a person has been washed overboard is printed. Line 5554 prints the new crew total, by subtracting 1 from CN, the variable representing the number of crew, but does not alter the value of CN at this stage. This is done during the end of the week report by checking if the strength rating of any member has been set to -999.

A crew member must, of course,be chosen to fall overboard. The selection is done in the loop at line 5558, which searches through the crew strength/type array, from the beginning, ignoring any elements set to 0 or -999, and selecting the first living member as the victim. Line 5560 dumps the person into the sea, by setting his strength rating to -999. The value of T is then set to 16, to make the program drop out of the loop.

The remaining events, dealt with in this module and labelled 2 to 5 in array RR(), are concerned with the four types of provision being washed overboard. To make the game more interesting, a random amount of each provision is selected for loss overboard. The amount of fruit, vegetables, meat and water is represented by PA(1), PA(2), PA(3) and PA(4).

In line 5572, 1 is subtracted from the random number, X, generated by subroutine 5500, because one event has already been covered by losing a crew member. The random number will now be from 1 to 4, corresponding to one of the four types of provision. Line 5574 checks if the provision is exhausted and takes no action if this is the case. The program then informs the player that some of the provision was washed overboard.

The amount of lost food or water is either a third, a quarter or a fifth of the stock. The fraction is picked randomly at line 5592, by dividing the amount of remaining provisions, PA(), by three, four or five, and subtracting that fraction from the original amount. The program then calculates the number of weeks' supply of that provision remaining using the equation at line 5594.

It is possible to run the program with the five contingencies mentioned. They all have a negative effect on the voyage, making it more difficult to succeed. Some positive contingencies will be inserted in the next instalment.

# Module Six: Random Events

### Initialise Events Variables

```
42 DIMRR(16)
43 REM INDICATORS TO SHOW IF RANDOM EVENT(N) ALREA
DY OCCURED
44 RC=0
45 REM COUNT OF RANDOM EVENTS SO FAR
46 RM=5
47 REM NO. OF RANDOM EVENTS IN PROGRAM
```

### Addition To Main Voyage Loop

```
860 GOSUB5500
861 REM GO TO GENERATE RANDOM EVENTS
```

### Random Events Subroutine

```
5500 REM RANDOM EVENT GENERATOR
5502 IFRC=RM THEN RETURN
5503 REM EXIT IF ALL RANDOM EVENTS DONE
5504 PRINTCHR$(147)
5505 GOSUB9200
5510 X=INT(RND(1)*RM)+1
5515 REM GENERATE RANDOM NO. BETWEEN 1 AND RM
5520 IFRR(X)=1 THEN RETURN
5522 REM RETURN IF THIS EVENT ALREADY DONE
5523 RR(X)=1:RC=RC+1
5524 REM SET IND. TO SHOW THIS EVENT DONE AND INCR
EMENT EVENT COUNT
5525 ON X GOTO 5540,5570,5570,5570,5570
5528 REM GO TO APPROPRIATE CODE FOR THIS EVENT
5530 PRINT:S$=K$:GOSUB9100
5535 GETI$:IFI$=""THEN5535
5539 RETURN: REM RETURN TO MAIN JOURNEY LOOP
5540 REM EVENT 1 - MAN OVERBOARD!
5542 PRINT
5543 S$="  DURING THE WEEK*":GOSUB9100
5545 PRINT:GOSUB9200
5546 S$=" 1 PERSON WAS LOST OVERBOARD*":GOSUB9100
5548 S$=" IN A STORM*":GOSUB9100
5550 PRINT:GOSUB9200
5552 S$="YOUR CREW IS NOW REDUCED TO*":GOSUB9100
5554 PRINTCN-1;"MEMBERS"
5558 FORT=1TO16
5559 REM SEARCH FOR CREW MEMBER TO LOSE
5560 IFTS(T,2)=0ORTS(T,2)=-999THEN5566
5562 TS(T,2)=-999:REM DEAD
5564 T=16
5566 NEXT
5568 GOTO5530
5570 REM EVENTS 2 TO 5 - PROVISIONS LOST
5572 X=X-1:REM X NOW POINTS TO PROVISION(1-4)
5574 IFPA(X)=0 OR PA(X)=-999 THEN5530
5576 REM NO ACTION IF THIS PROVISION ALREADY EXHAU
STED
5578 PRINT
5580 S$="  DURING THE WEEK*":GOSUB9100
5582 PRINT:GOSUB9200
5584 PRINT" SOME OF YOUR ";P$(X)
5586 S$=" WAS WASHED OVERBOARD*":GOSUB9100
5588 PRINT:GOSUB9200
5590 S$="YOU NOW HAVE APPROXIMATELY*":GOSUB9100
5592 PA(X)=PA(X)-INT(PA(X)/(INT(RND(1)*3)+3))
5593 REM REDUCE PROV AMOUNT BY 1/2 1/3 OR1/4
5594 PRINTINT(PA(X)/(CN*PN(X)));
5595 PRINT"WEEKS ";P$(X);" LEFT"
5599 GOTO5530
```

## Basic Flavours

**Spectrum:**
Make the following changes:

```
5504 CLS
5525 IF X=1 THEN GOTO 5540
5526 IF X>1 AND X<6 THEN GOTO 5570
5535 LET I$ = INKEY$ : IF I$ ="" THEN GOTO
       5535
```

**BBC Micro:**
Make the following changes:

```
5504 CLS
5535 I$ = GET$
```

# 'ZAPPING KLINGONS IN 17 DIMENSIONS'

**Because of PASCAL's enormous power and flexibility with regard to its data structures, such as records and sets, it needs to make much less use of arrays than other languages. In this instalment, we look at the use that PASCAL does make of arrays, including such aspects as packing, indices and the constraints made on dimensions.**

In many programming languages, we may find that the array is a very natural means of mapping real-world data such as lists, tables and matrices onto a computer data structure, and an efficient way of providing access to individual elements for processing. However, the main reason for their almost universal use is out of sheer habit. Quite simply, no data structuring method other than the array was available in most of the earlier languages. The earliest high-level language, FORTRAN, only had complex numbers and arrays, and its descendant, BASIC, omitted complex numbers. This is why the FOR . . NEXT loop is the only defined iteration structure.

That was fine in the days when FORTRAN was used only to perform matrix operations on every element of an array, and when BASIC was used only to introduce students to the rigours of programming in FORTRAN. We have already seen some examples of the extra flexibility of control that condition-driven loops (WHILE and REPEAT) bring to PASCAL. Perhaps even more importantly, we're beginning to glimpse some of the tremendous power and ease of expression that some of PASCAL's other data structures, such as sets and records, offer. The dominance of the array as a universal 'data tool' for programming is therefore considerably reduced in PASCAL.

## ARRAY FLEXIBILITY IN PASCAL

So although most programmers will feel on familiar ground when dealing with arrays in PASCAL, there are two significant points that should be kept in mind. Because PASCAL enables such a vast choice of data structuring methods, many programming problems may well be solved more effectively in terms of other structures. Secondly, arrays are entirely unconstrained in the structural complexity of their elements, so that much greater flexibility can be achieved in PASCAL.

The definition of an array type reserves storage for a certain fixed size of array, and defines both the base type of its index (subscript) and the type of each component. So, for example:

```
TYPE
  CharCounts = ARRAY [ 'A' . . 'Z' ] OF integer;
VAR
  list : CharCounts;
```

would reserve storage for 26 integers that would be referenced by specifying the array identifier followed by a bracketed indexing expression. Square brackets are always used with arrays, as they were originally in BASIC (the subsequent usage of round subscript brackets was intended to cater only for some limited character sets that lacked square brackets). To access a particular integer in the list just mentioned, we can write :

```
list [ 'M' ] or list [ pred (symbol) ]
```

In the second example, the expression pred (symbol) must, of course, evaluate to a char value in the subrange 'A' through 'Z'. Any true scalar type may be used to index an array dimension, but not reals or structured types. This disallows nonsensical attempts to refer to list [ 'second' ] or flag [ 3.75 ], for example. Using the illustrated type definition, we can initialise every counter element to zero in an array of type CharCounts with:

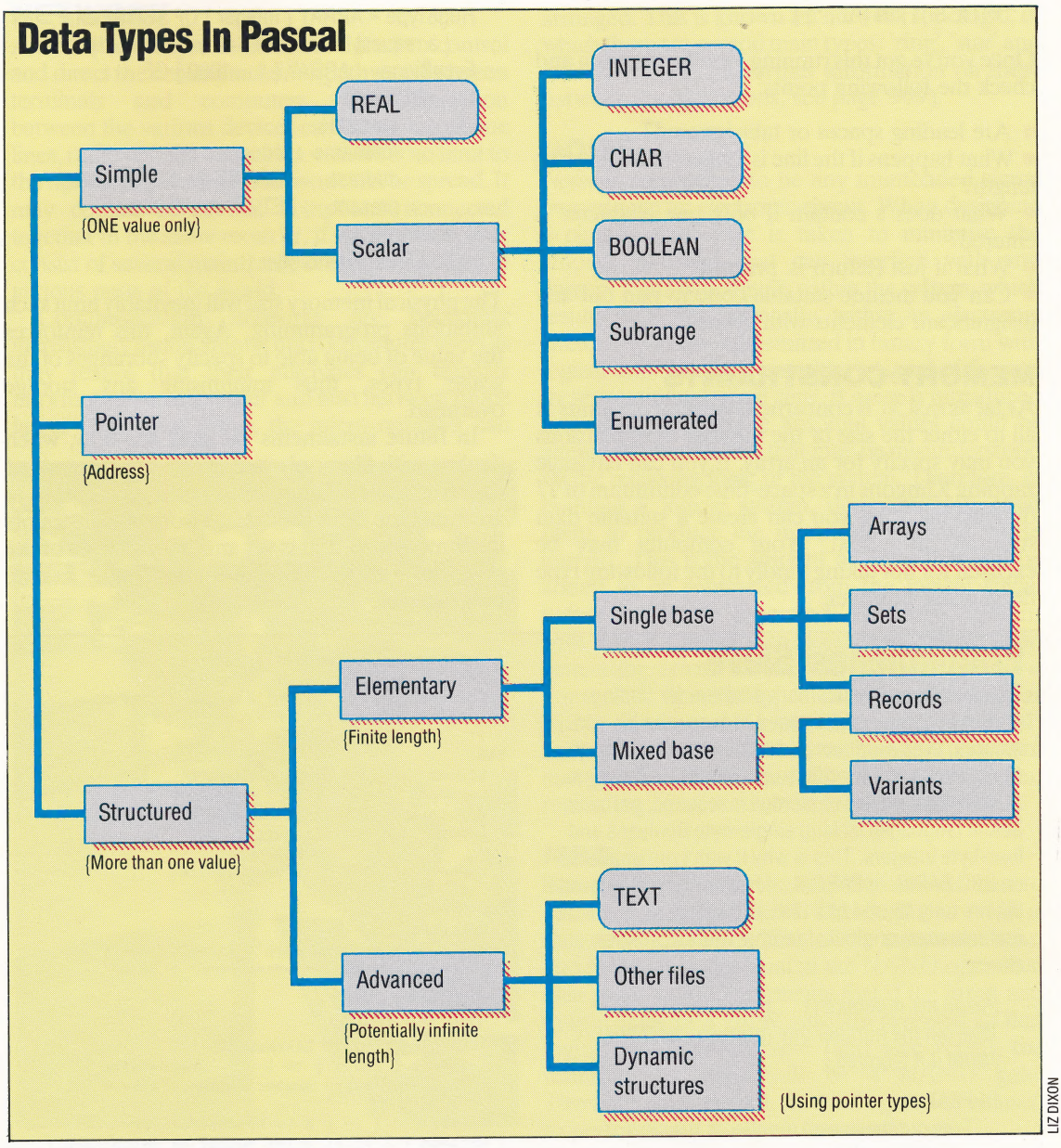
```
VAR
    letter  : char;
    counter : CharCounts;
BEGIN
  FOR letter :='A' TO 'Z' DO
    counter [letter] := 0
```

and so on. Obviously, counter [1] is illegal for this structure (it's the wrong index type) and counter ['a'] doesn't exist (the subscript is outside the defined subrange), so it's always good practice to define type identifiers for the indexing variables. In any case, we shall find that we invariably need them when defining our own procedures and functions.

If these subranges are in turn defined on values given in a CONST definition, the whole program may be rewritten to deal with different sized structures by the alteration of one line. For example, although there is no pre-declared type string in PASCAL, one way to model them (though not necessarily the best) would be :

```
CONST
  StringLength = 25;
TYPE
  StringSize = 1 . . StringLength;
  string = PACKED ARRAY [ StringSize ] OF char;
```

Notice that the reserved word PACKED can precede any structured type reserved word (SET, ARRAY, RECORD or FILE). By 'packing' data structures, we can gain considerable benefits in memory usage, particularly on computers with long word sizes (16/32 bits or more).




LIZ DIXON

**That's Typical!**
PASCAL's disciplined data structuring forces the programmer to adopt a more methodical approach to program construction. Only two of the three basic data types, shown here, can be subdivided into other categories. But the range of possible simple and structured data requires strict definitions of all the data used in a program. This extra effort often rewards the PASCAL programmer with more elegant solutions to programming problems than other languages can provide

## THE STRING TYPE

The only time you actually need to be aware of packing is when using literal string constants. A string of characters enclosed in quotes is taken to be indexed from one (not zero) and therefore declared implicitly as :

```
PACKED ARRAY [ 1 . . N ] OF char
```

in which N is the number of chars in the string. Consider this program:

```
PROGRAM PackString;
CONST
   Pascal ='Pascal';
TYPE
   string = PACKED ARRAY [ 1 . . 10 ] OF char;
VAR
   S          : string;
BEGIN
   S :=  Pascal;
   S := 'Too many characters';
   S := 'Pascal    '
END.
```

In the body of the program, the first two statements are illegal because of the incompatibility in lengths, but the third is acceptable (four spaces being used as padding characters). If the definition of the type string were not packed, all assignments to literals would be incompatible. In practice, these restrictions are not a great problem, however. Although PASCAL officially only supports the use of the read and write procedures for I/O of whole structures from and to files on backing storage, we can write literal or variable string types like these directly to the VDU. Try writing a program using a similar string declaration to the last example, and incorporate the statement:

```
REPEAT
   write ('String (Q to quit):');
   ReadLn ( S );
   WriteLn ('The string was :"',s,'"')
```

```
UNTIL S [1] IN ['Q', 'q']
```

Once you've got this running, test the program and check the following points:

- Are leading spaces or tabs ignored?
- What happens if the line is longer than the string length?
- What does s contain if only one character is entered?
- What if just Return is pressed?
- Can you include suitable code to pad out any insignificant elements with spaces?

## MEMORY CONSTRAINTS

As far as PASCAL is concerned, there are no limits at all to either the size or the number of dimensions you may specify for an array. If you can envisage zapping Klingons in a space-time continuum of 17 dimensions, then you can create a suitable data structure in PASCAL! Your computer may be excused for not taking kindly to the following type definitions, however:

```
HugeType = ARRAY [ integer ] OF SET OF char;
  { a request for at least 64K x 128 bytes!}
EvenBigger = ARRAY [ 1 . . 1000 ] OF
    RECORD
      surname,
      forename  : string;
      address   : ARRAY [ 1 . . 5 ] OF string;
      present   : SET OF attendances;
      {etc.}
    END: { EvenBigger }
```

The physical memory size will inevitably limit such ambitious programming. Again, this reinforces the value of being able to specify subranges of the scalar types, thus minimising any storage overhead.

In future instalments we shall see how, when dealing with files, only one or two components of the structure are required to reside in memory at any one time. This can free us from all constraints in memory size, and is yet another example of the powerful tools available to the PASCAL programmer.

## Eratosthenes' Sieve

This well-known benchmark program for finding primes is optimised for speed of execution by ignoring even numbers and using an array of 'flags' (8,192 elements for primes up to 16,384). Therefore, at least eight Kbytes of storage is required; perhaps 32K in languages lacking single-byte Booleans and four-byte integers. This is why it won't run unaltered on BBC BASIC. In PASCAL, however, the full set would require only 16,384 bits (2K) and enable us to model Eratosthenes' original algorithm more accurately and clearly:

```
Put all the numbers (1 . . max) into a sieve
Take out the number one (prime by defintion)
{ print it if required }

REPEAT
  Take out the lowest number in the sieve
  { print this prime }
  remove all its multiples from the sieve
UNTIL the sieve is empty
```

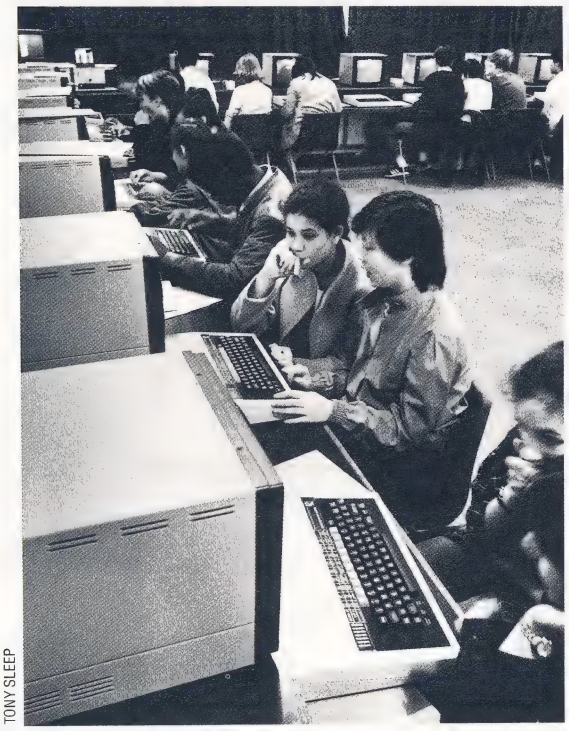
Such a large set will only be available on mainframes, but we can simulate one by using an array of smaller sets (100 or 1,000 elements, say, depending on your compiler). Each set's array index is found from the integer division of the number, and its membership of the set is represented by N modulo 100 or 1,000. Unless you have a 100-bit computer, however, the speed will probably be somewhat slower than the array version

```
PROGRAM     SieveList   ( output );
{ Find prime numbers by Eratosthenes' algorithm }
CONST
  SetSize        =   100; { ~ implementation }
  PredSetSize    =    99; { SetSize - 1 }
  MaxPrime       = 16383; { for MaxInts = 32767 }
  ListMax        =   163; { MaxPrime DIV SetSize }

TYPE
  PrimeRange     = 1 .. MaxPrime;
  dimension      = 0 .. ListMax;
  SetRange       = 0 .. PredSetSize;
  Siv            = SET OF SetRange;
  Eratosthenes   = ARRAY [ dimension ] OF Siv;
  cardinal       = 0 .. MaxInt;

VAR
  Sieves         : Eratosthenes;
  count,
  multiple       : cardinal;
  index          : dimension;
  N              : PrimeRange;
  number         : 0 .. PredSetSize;

BEGIN
  WriteLn;
  WriteLn ( 'Eratosthenes'' Sieve' : 50 );
  WriteLn ( '==================' : 50 );
  WriteLn;
  WriteLn;

  FOR index := 0 TO ListMax DO
    Sieves [ index ] := [ 0 .. PredSetSize ];
        { put every number in the sieves }

  Sieves [ 0 ] := [ 2 .. PredSetSize ];
{ WriteLn ( 1 ); }{ prime number by definition }
  count := 1;

  FOR N := 2 TO MaxPrime DO
    BEGIN
      index := N DIV SetSize;
      number := N MOD SetSize;

      IF number IN Sieves [ index ] THEN
        BEGIN
          count := succ ( count );
        { WriteLn ( N ); }
          Sieves [ index ] := Sieves [ index ]
                                - [ number ];
          multiple := N + N;

          WHILE multiple <= MaxPrime DO
            BEGIN
              index := multiple DIV SetSize;
              Sieves [ index ] := Sieves [ index ]
                        - [ multiple MOD SetSize ];
              multiple := multiple + N
            END
        END
    END;

  WriteLn;
  WriteLn ( count : 25, ' primes found.' )

END .
```

## NETWORK

A *network* is a system that is designed to control and direct the communications pathways between terminals and computers. Communication between the various devices can be via telephone lines, radio or infra-red waves — there is no limit to the distance over which a network can be spread. It may consist of several computers connected together in the same room or, if via satellites, may consist of several mainframe computers placed in various parts of the world.

Communication networks are becoming increasingly useful among microcomputer users as they realise that their machines can transfer information to each other and also between much larger computers. Thus, the micro operator can take advantage of the much larger processing power — as well as the enormous databases — available on mainframes.



TONY SLEEP

**Share And Share Alike**
Networks are increasingly being used in education, because they not only allow information held on one computer to be transferred to another, but also enable scarce resources to be shared. Shown here is a school classroom using the BBC Micro's Econet system

The term network is also used to refer to the interconnected components of an electronic circuit. This definition of the word has several subdivisions. An 'active network' contains amplifying and logic devices such as transistors, while a network without any of these is said to be 'passive'. Similarly, a 'linear network' contains no non-linear devices such as diodes, whereas a 'non-linear network' will.

## NETWORK ARCHITECTURE

This refers to the way in which a network has been designed and configured. Specifically, areas covered by *network architecture* consist of transmission codes including error checking and data flow control, the way data is transmitted, and the method by which terminals are addressed within the network. Network architecture also concerns itself with the way the network is arranged. This is known as 'network topology', of which there are several main types: 'ring', 'star' and 'bus' networks (for further information on these network configurations, see page 969).

## NOISE

*Noise* is considered to be any unintended signal occurring in an electronic circuit. When designing a circuit, great care is taken to minimise the amount of noise that can interfere with and corrupt legitimate signals within the system, which results in errors. Typically, within an electronic circuit, signals are transmitted in binary form with logical one being generated by a signal of 5v, and 0v representing logical zero. If noise is introduced into the system, these absolute values could be altered to other values that may be misunderstood by the receiving logic of the system. Therefore, in building a logical system, designers take great pains to produce the maximum possible 'noise margin'. This is the amount of noise that can be added to or subtracted from the logical signal before errors are generated. Electronic systems with a high tolerance to noise before they begin generating errors are said to have a high 'noise immunity'. Specific types of noise that appear within circuits include 'impulse noise' (page 794) and 'white noise': the latter being of constant energy over equal periods of frequency, and remaining continuous in amplitude and time.

## NOR

This is one of the six Boolean expressions used to construct logic structures. The output of *NOR* is said to be true when both of the inputs are false, and is said to be the dual of the NAND operation (see page 1128). This means that if the input and output lines of a NOR gate are reversed so that logical one becomes zero (and vice versa), the resulting operation will be a NAND gate. Therefore, like its dual, the NOR gate can be used to perform any Boolean operation.

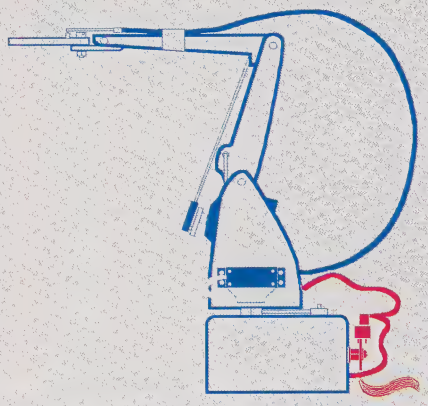
## NOT

*NOT* is another of the Boolean operations. The result of performing this operation is that the output value of true or false will be the opposite of the input value. Thus, if the input is true then the ouput is false, and vice versa. Note that unlike many of the Boolean expressions in which there is a pair of inputs, the NOT operation has just one.

## NUMERICAL ANALYSIS

A branch of mathematics that is concerned with solving problems by numerical methods, *numerical analysis,* in simple terms, means that problems are solved by 'number crunching'. Because it usually involves large quantities of numbers and iterative methods, it is ideally suited to the computer. However, due to the large amount of material that has to be processed, numerical analysis concerns itself with the most efficient and suitable method with which to solve the widest possible number of applications.

# FULLY ARMED



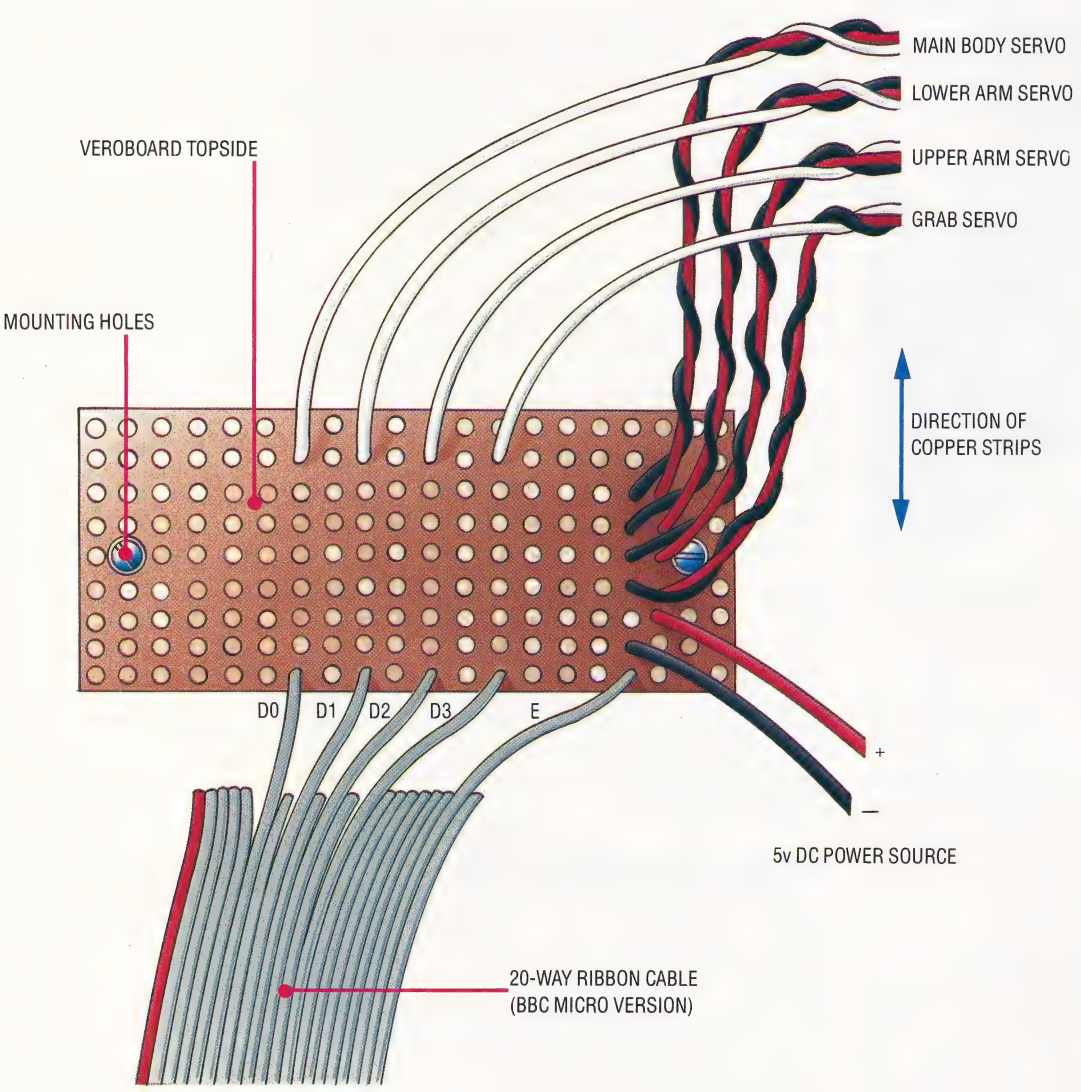
**Having completed the major construction tasks for our Workshop robot arm, we need only make the electrical connections to finish this process. The four servo motors are provided with power and control lines, and this will enable us to begin developing the software.**

### Step 1: The Veroboard Connections

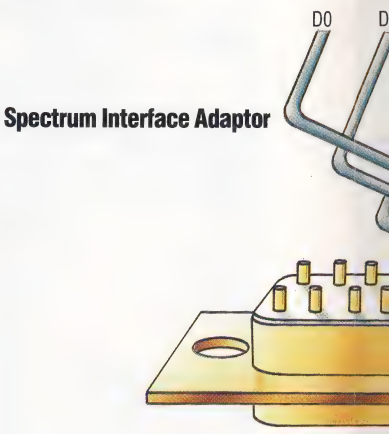
Each servo motor has three wires attached to it. On the Futaba servos recommended for this project, these wires are coloured white for the control line, red for the positive power line and black for the common return line. The servo wires must be linked to the controlling computer and power source using a 20-strip by nine-hole rectangle of veroboard, which will be mounted on the back of the arm's base box. Cut the veroboard to size and drill two mounting holes in the positions shown. Use the veroboard as a template to mark hole positions on the back of the base box and drill out. The holes in the base box and veroboard should be drilled so that a machine screw can be pushed through. Drill a further hole in the back of the base box and push through the group of wires from the servo mounted within the base.

Gather together the four groups of wires from the servos to the back of the base box and solder the wires in the positions shown. The four data lines and computer earth should be soldered onto the bottom of the verobard. The diagram shows a 20-way ribbon cable used with the

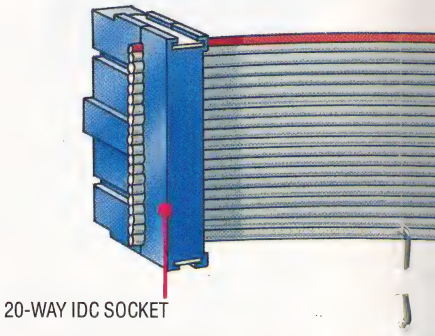
## Step 1: The Veroboard Connections



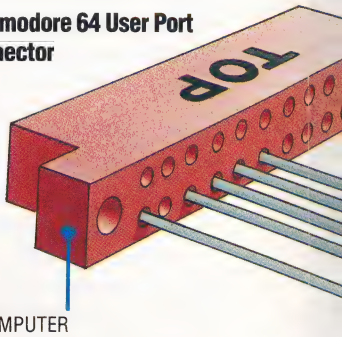

## Step 2: Computer Po




**BBC Micro User Port Connector**




**Commodore 64 User Port Connector**



BBC Micro. Spectrum and Commodore 64 versions use five separate wires, or a 5-way piece of ribbon cable. Notice that the white control lines from the servos solder into positions directly above each data line from the computer and that the black lines all solder into a single copper strip above the computer earth connection and the two power supply connections.

The final soldering task is to attach a 5v power source to the board as shown. The 5v supply available from the BBC Micro, Spectrum or Commodore 64 is not suitable to power all four servos under a heavy load. An alternative source should therefore be sought. A 4.5v battery (or three 1.5v batteries in a battery clip) makes an ideal power source. If a 5v DC transformer is available, then this too could be used. If you intend to use batteries, then a battery clip should be soldered onto the free ends of the power lines extending from the veroboard. If a transformer is to be used, a suitable in-line power connector should be attached. The negative side of the power source shares the same copper strip as the computer earth and the black return lines from the servos; the postitive power lead connects to each of the red wires from the servos via a common copper strip.

### Step 2: Computer Port Connections

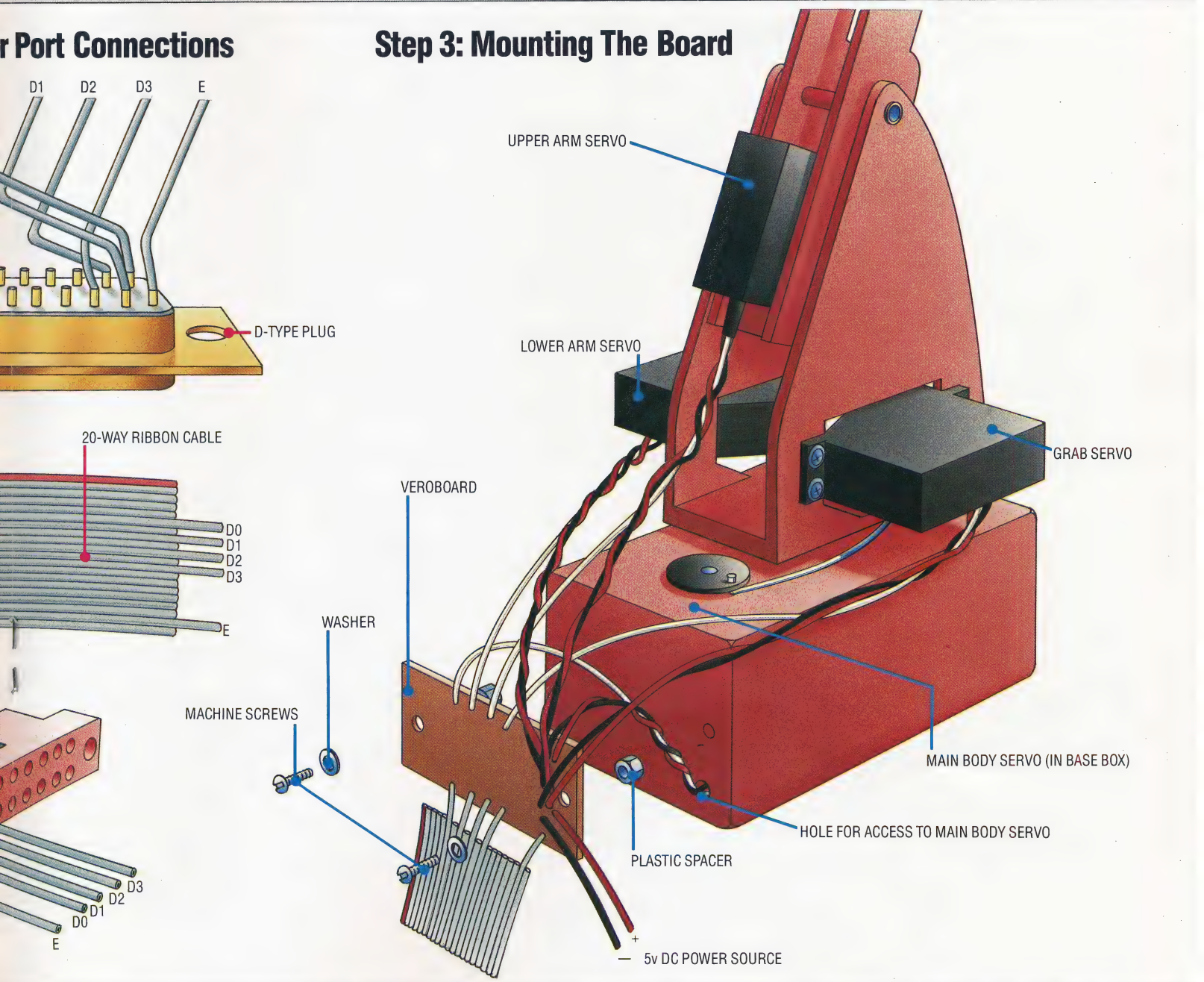
Having completed the veroboard wiring, the appropriate computer port connector must be added to the free ends of data lines D0 to D3 and the earth line. For the BBC Micro, a standard 20-way cable and IDC socket should be used. The Commodore 64 uses a 24-way 0.15in edge connector. Because this connector plugs into the user port either way up, mark one side as TOP before starting. The five lines from the veroboard should be soldered as shown.

Both the BBC and Commodore 64 micros have on-board circuitry to handle control applications via a user port. The Spectrum, however, has no such circuitry and a special interface must be built to plug into its expansion port. We have already designed and built such an interface in previous instalments of Workshop (see pages 992, 1012 and 1034). Spectrum owners should now refer to those instalments to build the interface.

Data and power lines from the interface are provided by a 12-way D-type socket that plugs directly into the Workshop robot. We can adapt this connector for use with the robot arm. Wire up a 15-way D-type plug (Maplin Part No. BK58N) using the free ends of the data and earth lines from the arm's piece of veroboard and attach a plug cover (Maplin Part No. BK60Q). This adaptor allows the robot arm to plug into the Spectrum's expansion port via the interface board.

### Step 3: Mounting The Board

When all the wiring has been completed and checked carefully, the veroboard can be mounted on the back of the base box using two machine screws fastened with nuts inside the base. Ensure that the motors connect to the veroboard in the correct order. Data line D0 (the furthest to the left) should connect to the white line of the main body servo, mounted in the base box; D1 should connect to the lower arm servo, mounted on the left of the main body; D2 controls the upper arm servo, mounted near the shoulder joint, and D3 controls the grab servo, mounted on the right of the main body.




KEVIN JONES

# MAKING RECORDS

**Many database managers possess a built-in language allowing you to set up searches and access information with the minimum of effort. Using examples from Ashton Tate's dBase II, we will see how powerful and efficient a DBM can really be as we create and enter data into simple files.**

There's more to databases than just organising data as a collection of records in a file. Accessing records is possible using simple commands, but the better DBMs allow users to write procedures using a built–in programming language. This can extend the power and usefulness of a database considerably.

To understand the use of user-written procedures in databases, we should first review the way most DBMs use commands. A command, such as SEARCH, LOCATE, DISPLAY, NEXT, LAST and so on, is processed by the DBM to find out what the user wants. Having done that, it opens the database file and goes through it sequentially, starting with the first record and working its way through to the last record, extracting en route the records that meet the requirements. Such commands are sometimes said to belong to a 'query language'.

In contrast to direct commands or queries, many DBMs — Psion's Archive (see page 1124) and Ashton Tate's dBase II are two good examples — allow programs or procedures to be written by the user to simplify the process of extracting or collating information. These procedures are more than just collections of standard commands or queries. They are written in complete (if limited) programming languages that are built in as part of the DBM. Some of these DBM programming languages are quite similar to ordinary programming languages, and are quite easy to learn. Psion's Archive incorporates a language that is, to all intents and purposes, identical to Sinclair's SuperBASIC. Writing procedures to use with Archive is perfectly straightforward for anyone familiar with SuperBASIC.

Although the dBase II language is not quite like any other language, it is sufficiently like a structured BASIC with a few reserved words to be easy to learn and use. Its structures are fairly simple, and include DO WHILE . . . ENDDO, DO CASE . . . ENDCASE and IF . . . ENDIF. In addition, there are about a hundred commands — some familiar and some tailored specifically for use within a database environment. Familiar-looking commands include CALL (to call a machine language), NOTE (equivalent to REM in BASIC), STORE <expression> to <variable> (equivalent to LET <variable> = <expression> in BASIC) and many others. Less familiar commands include SKIP (to move forward or backward through records in the file), PACK (to remove unwanted records from the file) and FIND <character string>.

Several functions are also provided, including familiar, BASIC-like functions and more unusual ones. Examples include CHR <expression> (equivalent to CHR$(x)), LEN <character string expression> (equivalent to LEN in BASIC), TYPE <expression> (returns the type of expression) and DATE() (a system variable containing, naturally enough, the date).

To see how useful a custom-written procedure can be, compared with simply using a sequence of commands from the keyboard to extract information, we'll use dBase II to create a stock and inventory database, and then write a simple procedure in the dBase II language to extract some information. The records in the file will be quite simple, with four numeric fields and one character field. A typical record (also used in previous instalment) looks like this·

```
MAKERSPART NUMBER                    06116
OURPART NUMBER                        3995
NUMBER LEFT IN STOCK                    86
PRICE                                34.75
DESCRIPTION          Dongle with widget nozzle
```

First we need to find a one-word name for each field, and to decide on the lengths of the fields, and whether they are to be 'character', 'numeric' or 'logical' fields. This should be sufficient:

```
MAKERSPART  : 5 characters; numeric
OURPART     : 5 characters; numeric
REMAINING   : 3 characters; numeric
PRICE       : 6 characters; numeric
DESCRIPTION : 40 characters; character
```

Once dBase II is running, it responds with a full stop (period) prompt on the screen that tells you it is waiting for input. The command that starts a new file is CREATE, and we will also have to supply a name for the file as a whole (we'll use PARTS as the filename). The process would look like this:

```
. create parts <CR>
ENTER RECORD STRUCTURE AS FOLLOWS:
FIELD   NAME, TYPE, WIDTH, DECIMAL PLACES
 001    makerspart,n,5,0
 002    outpart,n,5,0
 003    remaining,n,3,0
 004    price,n,6,2
 005    description,c,40
 006    <CR>
```

Our input has been given in lower case letters and dBase II's response has been printed in upper case letters. Notice that numeric fields require the number of decimal places to be specified as well as the width of the field. Pressing the Return key without entering any information, as in field 006, terminates the initial phase of file creation.

Next we need to enter some data. This is done by using the APPEND command. This causes the screen to clear and a 'skeleton' of the record to appear, waiting for data to be entered.

```
MAKERSPART  :          :
OURPART     :          :
REMAINING   :      :
PRICE       :          :
DESCRIPTION :
```

Data can now be typed in at the keyboard and will be fed to each field in turn. A carriage return signals that entry to a field has finished; as soon as Return is pressed for the final field, all the data will be accepted and the skeleton record will reappear, waiting for details of the next record. Terminating data entry is done by pressing Return at the start of an empty record.

Having entered data into the file, you will want to use it. Suppose that, for auditing purposes, you needed to know the retail value of your entire stock. One way this could be done would be to look at each record in turn on the screen and make a note of the number of parts remaining in stock for each item, record the price of each and multiply the two figures. After looking at all the records and noting all the prices and numbers of each item in stock, you could then get the total value by multiplying each price by the number of parts and then adding all the subtotals. If you were using a conventional card index, this would indeed be the only way of arriving at a solution.

Most DBMs allow this sort of information to be retrieved far more simply. Here's how you would do it with dBase II:

```
. use parts
. sum remaining * price
37870.58
```

The first line is a command instructing dBase II to work with the file called PARTS. The second line is a command meaning 'add together for all records the result of multiplying the REMAINING field by the PRICE field of each record'. The third line is the sort of response that might be returned after the total value, in pounds, of the entire stock. Beats card indexes, doesn't it?

This example clearly illustrates how powerful a good DBM can be, even using nothing more than a simple command. But the real power comes from storing sequences of commands as a program. To store our simple stock valuation commands as a program called VALUE, the following dialogue would be needed:

```
. modify command
ENTER FILE NAME: value
set talk off
use parts
sum remaining * price
set talk on
cancel
```

This short program will be stored on disk and can be used from within dBase II at any time by typing in the command:

```
. do value
```

The program will be automatically read in and executed, and as soon as it has finished, control will be returned to dBase II.

## Accounting Procedure

```
* Display Accountants
SET TALK OFF
USE MEMBERS
DO WHILE .NOT. EOF
   IF JOB = "ACCOUNTANT"
      DISPLAY NAME
   ENDIF
   SKIP
ENDDO
```

The program we give here, which extracts the names of club members who are accountants, is written in Ashton-Tate's dBase II command language. The first line, beginning with an asterisk, indicates that this is a comment line. We do not require that all the record numbers be displayed, as dBase II carrys out the search, so we SET TALK OFF.

There may be a number of different files held on the database, and so we must specify which file (in this case MEMBERS) is to be searched. As we wish dBase II to search through all available records, we construct a loop around the DO WHILE . . . ENDDO structure.

Within the loop we set a condition that states whenever the job field contains the string 'ACCOUNTANT', the program should display the record's name field. Note that the IF condition must, of course, be terminated by ENDIF. The program then drops through to the SKIP command which tells dBase II to go on to the next record


**Discovering New Worlds**
The Discovery 1 is intended to be an all-in-one expansion system for the Spectrum user. The machine comes complete with a disk drive, operating system and connections to allow printers, joysticks, composite video monitors and other peripherals to be fitted without the need for further interfaces

# OPUS SESAME!

**Sinclair Research has provided the Microdrive for the Spectrum, but a need has remained for a more reliable and accommodating disk drive system. We conclude our two part series on alternatives to the Sinclair Microdrive by examining the Discovery 1 from Opus, following our look at Rotronics' Wafadrive (see page 1129).**

In the previous Hardware article we looked at the Rotronics Wafadrive. Although this device proved to be somewhat more reliable than the Microdrive, it turned out to be slower in comparison, and still based around the rather suspect continuous tape system. In this article we turn our attention to a more conventional approach to mass storage: a disk-based system from Opus Supplies.

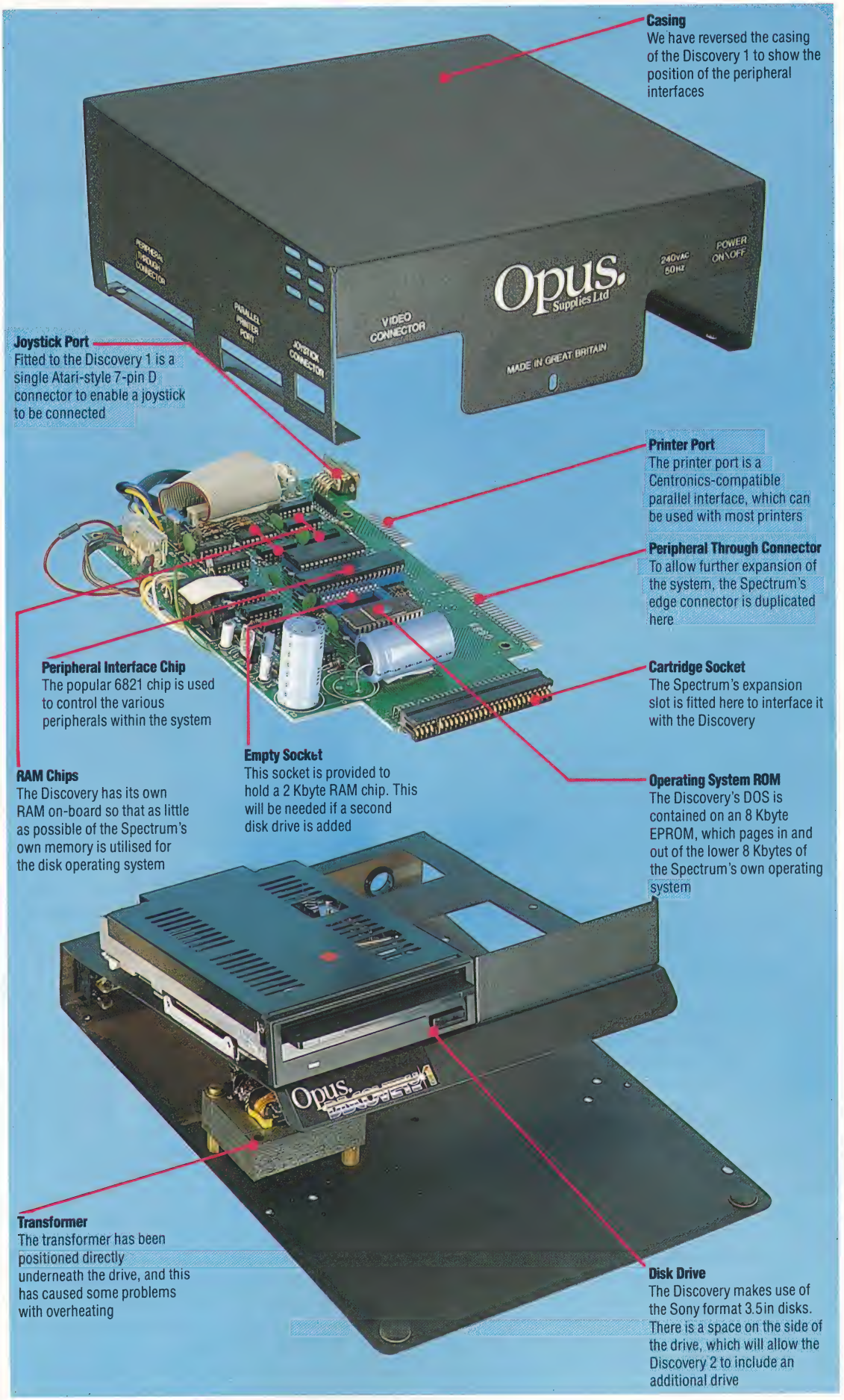
A disk drive for the Spectrum does not represent a new idea. Over the past couple of years there have been a number of disk operating systems and interfaces available for the machine; however, none of these systems have become particularly popular. This is partly because the interfaces have been advertised solely in the specialist press, which has given the devices an air of being intended only for the devout Spectrum enthusiast and machine code programmers, and partly because they have been available only through mail order houses and have not been given the kind of mass marketing needed to push them into the public consciousness.

The Discovery 1 is encased in sheet metal, extending the front of the machine to support the Spectrum that plugs into a cartridge slot via the expansion port. Although this seems a comparatively easy operation, users may find some difficulty in attaching the Discovery to the edge connector. This is because the cassette and aerial leads tend to get in the way and prevent the cartridge slot from being attached correctly. Bearing in mind that a badly attached edge connector could severely damage the Spectrum, this is a serious problem. The difficulty is not as pronounced with the original Spectrum (for which the Discovery was originally designed), but the new Spectrum+, with its larger casing, may have users struggling for several minutes before they are satisfied that the device is correctly fitted. Once the Spectrum has been fitted to Discovery 1, the machine effectively blocks off the computer's own power supply input. The Discovery has therefore been designed to power both itself and the Spectrum, thus making the micro's own external power supply redundant.

## DUAL DRIVE UPGRADE

Above the cartridge slot on the left is a single 3½in disk drive, with space on the right for a second drive. (Opus intends to launch a dual drive version of the machine called the Discovery 2.) Discovery 1 users wishing to upgrade their systems to dual drive configurations can expect an external additional disk drive called Discovery+ to be launched. However, users will not be limited only to Opus drives since the company claims that standard 5¼in disk drives can also be added in the same manner.

In common with Rotronics, Opus's philosophy in designing the machine is to provide not only a




**Casing**
We have reversed the casing of the Discovery 1 to show the position of the peripheral interfaces

**Joystick Port**
Fitted to the Discovery 1 is a single Atari-style 7-pin D connector to enable a joystick to be connected

**Printer Port**
The printer port is a Centronics-compatible parallel interface, which can be used with most printers

**Peripheral Through Connector**
To allow further expansion of the system, the Spectrum's edge connector is duplicated here

**Peripheral Interface Chip**
The popular 6821 chip is used to control the various peripherals within the system

**Cartridge Socket**
The Spectrum's expansion slot is fitted here to interface it with the Discovery

**Empty Socket**
This socket is provided to hold a 2 Kbyte RAM chip. This will be needed if a second disk drive is added

**RAM Chips**
The Discovery has its own RAM on-board so that as little as possible of the Spectrum's own memory is utilised for the disk operating system

**Operating System ROM**
The Discovery's DOS is contained on an 8 Kbyte EPROM, which pages in and out of the lower 8 Kbytes of the Spectrum's own operating system

**Transformer**
The transformer has been positioned directly underneath the drive, and this has caused some problems with overheating

**Disk Drive**
The Discovery makes use of the Sony format 3.5in disks. There is a space on the side of the drive, which will allow the Discovery 2 to include an additional drive

| DISCOVERY 1 |
| --- |
| **PRICE** |
| Discovery 1, £199.95; Discovery + upgrade £139.95; Discovery 2 £329.95; all including VAT |
| **DIMENSIONS** |
| 300×210×75mm |
| **INTERFACES** |
| Parallel through connector, Centronics parallel interface, joystick port, composite video jack |
| **FORMAT** |
| Single-sided double-density 3½in Sony standard disks |
| **CAPACITY** |
| 250 Kbytes total, 180 Kbytes formatted |
| **SPEED** |
| Transfer rate 15 Kbaud, track to track access time 3 milliseconds |

mass storage system for the Spectrum, but also to add extra peripheral interfaces allowing users to run printers and other devices. A composite video monitor socket on the back of Discovery 1 has been provided, according to an Opus spokesman, for business users wishing to attach a monochrome monitor (although of course composite video does produce a colour signal) for lengthy periods of word processing. However, on a machine noted for its colourful games programs, it is a pity that Opus could not have provided an RGB interface to produce a much clearer picture.

On the right side of the Discovery is a single Kempston–compatible, Atari–standard joystick port, next to which is a bi-directional Centronics parallel printer port. Finally, there is a peripheral through connector to enable other Spectrum-compatible interfaces, such as an RGB monitor, to be connected.

Like the Wafadrive, the Discovery disk operating system closely follows that of the Interface 1; for example, issuing a command requires <COMMAND> *. When the BASIC interpreter reaches the *, it does not recognise it as being a BASIC command and attempts to generate a syntax error. However, the DOS intercepts it and pages its own eight Kbyte operating system into the position of the lower eight Kbytes of the Spectrum's ROM, and then interprets the command. It should be noted that if the user has made a syntax error, even the DOS won't recognise it and an error message will be generated, although this will still be via the DOS ROM.

In designing its DOS system, Opus has gone further than Rotronics in providing compatibility — all of the commands available to the Microdrive have been retained. There are several reasons for this. Because of the Spectrum's single keyword entry system, it is obviously easier to write an operating system using inherent commands, rather than going to the trouble of writing your own. This also means that users who are already familiar with the Microdrive operating system will be able to use the Discovery immediately, since all the syntax is the same. Furthermore, tinkering with an operating system can lead to all kinds of unforeseen problems with the memory map. This means that programs compatible with Interface 1 may not necessarily be compatible with your revamped operating system, a problem that has plagued many other third party peripherals.

The way in which Opus has closely followed the Sinclair Microdrive command system is most noticeable when looking at the way the streams have been organised. On the Spectrum, output channels are organised into 16 streams numbered 0 to 15. Three of these are set aside for the screen, keyboard and printer, the others are free for use by any other peripheral. In the Sinclair list of Interface 1 commands, there are a number of single characters that open channels to specific devices; for example 'm' for Microdrive. The Discovery has adapted these to its own use, thus the command LOAD * 'm';1;'name' will work just as well on Discovery 1 as on a Microdrive, although in this case 'm' refers to the disk drive. However, for added convenience, Opus has adapted the command format so that 'm' can be omitted, thereby shortening the somewhat long-winded Sinclair system. Other commands have also been adapted. The character 't' in a command on the Microdrive opens a channel to the RS232 interface, whereas on the Discovery it opens the channel to the parallel printer.

## THE DISKS IN OPERATION

The disk drive supplied with the system uses the double–density Sony format 3½in disks that are becoming increasingly popular on microcomputers. The disks themselves each have a total capacity of 250 Kbytes, which, when formatted, provide 180 Kbytes of available storage space. The disk operating system supports random access when searching for a file, which is considerably faster than the serial search methods used on some other disk systems. Also, there is no limit to the number of files that can be held on a disk, which can be important when one wishes to save a number of short files. If the directory is quickly filled, there may be a large amount of space on the disk that cannot be used.

When comparing the time it takes to SAVE and LOAD a file using the Discovery and Microdrive, the former proved to be somewhat faster in actually finding a file but considerably slower in SAVEing and LOADing it. Finding a file is faster on the disk system because the files are organised by random access whereas the Microdrives, by their nature, are serial access devices. Why accessing a file into memory is much slower is more difficult to explain, but it is a fact that the transfer rate of the Discovery 1 is much slower than that of the Microdrives — 15 Kbaud compared to 19.2 Kbaud. The real advantage of the Discovery lies in having a mass storage system more robust than the Microdrive's, and a storage medium having a wider range of manufacturers to choose from.

Opus appears to have given some thought to the problematic aspect of software support for the Discovery. Obviously, having a large company like Boots selling the product in their chain of stores is an advantage, since software houses will be able to offer their products alongside the machine itself. The company has also indicated that many software houses, including Melbourne House and Legend, have already agreed to transfer some of their existing programs onto the 3½in disks.

The launch of the Discovery series of disk drives has clearly been well planned and Opus has obviously attempted to provide its new line with as much chance of success as it possibly could. For the company, the major task before it is convincing Spectrum owners that the Discovery is a more worthwhile investment for their machine than the Sinclair alternative. If the time is right and the Spectrum owner is ready for a disk–based system, Opus and its Discovery 1 could well be a success.

# PORTS OF CALL

**We begin a two-part investigation of how the Commodore 64's operating system manages input and output. Here, we take a detailed background look at the peripheral ports on the computer, give a brief description of how the kernel I/O routines can be accessed and show how OS routines can be used to drive RS232.**

Our photograph shows all the available ports at the back of the Commodore 64. As well as the cassette port, the audio/video port and the TV socket, there are three I/O ports available to the programmer. These are, from left to right, the expansion port (also known as the cartridge port), the serial port (or serial bus) and the user port. We will look at each of these separately.

- *The Serial Port:* Commodore serial printers and the 1541 serial disk drive plug into this six-pin DIN socket. The OPEN statement will always apply to this port with any device number other than 2 (specifying device 2 means RS232 via the user port). For example, with device number 8, the command OPEN2,8,2,"FILENAME" will open a file to the disk drive through the serial port. It is not advisable to attempt to use the serial port, except with Commodore devices, through BASIC or the kernel routines.

There are interfaces available, however, which accept serial messages via this port and perform a serial/IEEE parallel conversion. This enables peripheral devices that interface with the Commodore PET to be used on the Commodore 64. Such peripherals include the Commodore 4040 disk drives.

- *The User Port:* This a flat 24-way, 0.15in pitch male edge connector, which can be used for both parallel and serial communication. For example, this port can be used to connect the Commodore 64 to a non-Commodore printer (Epson is a popular alternative), which is treated as an RS232 device. We will consider how RS232 can be driven from the OS later in this instalment.

The user port can also be used for eight-bit parallel communication, but you have to write — or obtain — your own device-driving software to use it in this way, since the OS has no 'driver' routines (machine code programs that control peripherals) for parallel communication. In the next instalment we provide a program called Parawedge, which illustrates the amount of work required to provide a moderately sophisticated, interrupt-driven parallel communications program. This could be used to transfer a block of memory from the computer to an external device (such as a BASIC program from one Commodore 64 to another), provided that both machines have 5v levels.

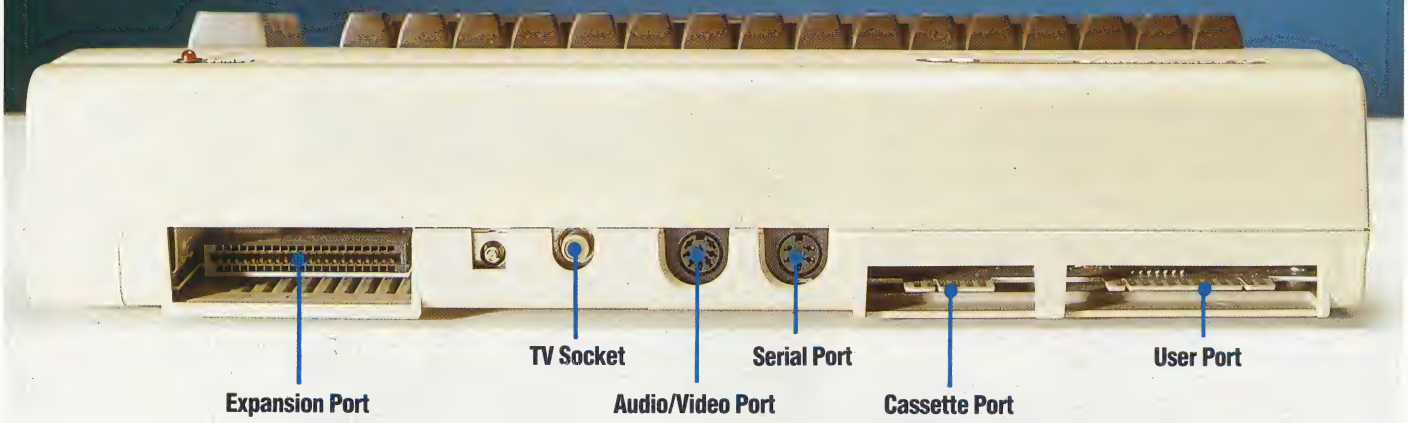
- *The Expansion Port:* The expansion port is a 44-pin female edge connector that gives access to all the main control, address and data lines on the Commodore 64. This port is used to interface games cartridges and parallel IEEE cartridges, which allow the computer to be connected to PET peripherals. It is also used for any other devices or software that require almost complete control of the machine, or programs sufficiently good to warrant the design and construction of a card to hold them as ROMs or EPROMs.

We give diagrams for each of these three ports, showing the functions of their pin connections. Now let's turn our attention to aspects of the operating system's use of these ports. The set of OS routines that deal with I/O is known as the kernel. These are the machine code programs called by the BASIC I/O statements OPEN, CLOSE, GET#, PRINT#, and so on. Commodore have arranged that the kernel routines are easily accessed by the machine code systems programmer. We give a short machine code listing that allows us to use the kernel LOAD routine.

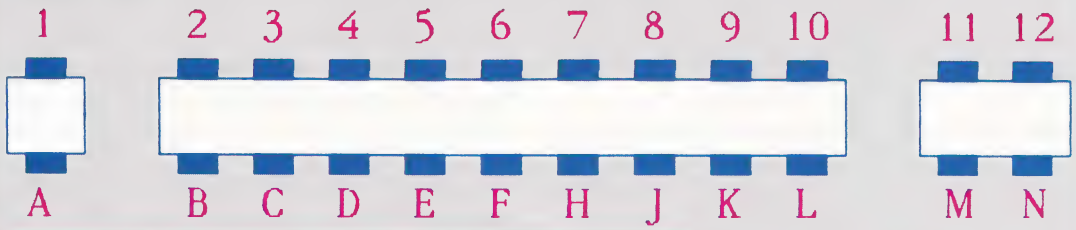
We also examine here the Commodore 64's built-in RS232 capability, and consider how this can be employed, for example, to drive a modem via the RS232 routines.

It won't always be possible, however, to use the system routines for I/O. Sometimes you may need to use parallel communication and high data rates to a nearby device of your own, rather than serial communication to some more distant device. There are no system routines for driving the user port on the Commodore 64. So parallel communication means programming the two 6526 Complex Interface Adaptors (CIAs) yourself. To get you started on this an eight-bit parallel data transfer assembler routine will be given in the next instalment. This routine enables the computer to read or send data via the user port, while at the same time processing a BASIC program. The magic of time-sharing is accomplished by arranging that the code that reads or writes data is interrupt-driven. In the previous article in this series, we gave an example of an IRQ — interrupt request — wedge (see page 1137). In the present case the routine is NMI-driven because the user port flag line can be used to generate an NMI (non-maskable) interrupt: this is the second interrupt line to the 6510 processor. Unlike IRQ, an interrupt signal on the NMI line cannot be masked out using the SEI and CLI commands.

# Commodore 64 I/O Ports




## User Port



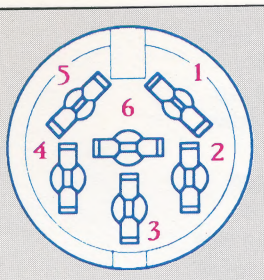

| Top | | |
|---|---|---|
| **Pin** | **Type** | **General Purpose** |
| 1 | GND | Ground |
| 2 | — | +5 volts at 100 mA maximum |
| 3 | — | System RESET |
| 4 | CNT1 | CIA #1 serial port counter |
| 5 | SP1 | CIA #1 serial port |
| 6 | CNT2 | CIA #2 serial port counter |
| 7 | SP2 | CIA #2 serial port |
| 8 | PC2 | Handshaking line from CIA #2 |
| 9 | — | This line is connected to ATN on the serial port. |
| 10 | — | 9v ac +phase |
| 11 | — | 9v ac -phase |
| 12 | — | Ground |

**Notes:**

1) 3-line = Xon/Xoff protocol
X-line = CTS/RTS protocol
2) (*) In addition to being addressed via the OS RS232 kernel routines, these lines can also be programmed directly for user defined I/O. These connections were used to contol external devices such as the buffer box and floor robot in our Workshop series

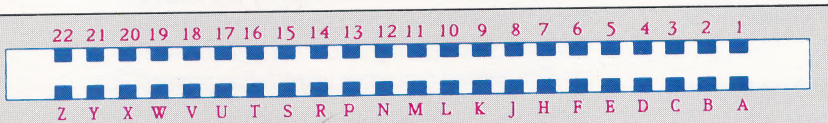
| Bottom | | | |
|---|---|---|---|
| **Pin** | **Type** | **RS232 Function (*)** | **Used In 3-Line/X-Line** |
| A | GND | Protective ground | Both |
| B | FLAG2 | Received data —in | Both |
| C | PB0 | Received data —in | Both |
| D | PB1 | Request to send (RTS)—out | Both (high in 3-1) |
| E | PB2 | Data terminal ready (DTR)—out | Both (high in 3-1) |
| F | PB3 | Ring indicator (RI )—in | — |
| H | PB4 | Received line signal (DCD)—in | X-line only |
| J | PB5 | Unassigned | — |
| K | PB6 | Clear to send (CTS)—in | X-line only |
| L | PB7 | Data set ready (DSR)—in | X-line only |
| M | PA2 | Transmitted data —out | Both |
| N | GND | Signal ground | Both |

## Serial Port




| Pin | Type | Comments |
|---|---|---|
| 1 | SERIAL SRQIN | Request service |
| 2 | GND | |
| 3 | SERIAL ATN I/O | Attention signal to devices on port |
| 4 | SERIAL CLK I/O | Timing clock for serial port |
| 5 | SERIAL DATA I/O | Single bit data transfer line |
| 6 | RESET | Hardware reset line |

## Expansion Port




**Top**

| Pin | Type | Comments |
|---|---|---|
| 1 | GND | |
| 2 | +5v | |
| 3 | +5v | |
| 4 | IRQ | Interrupt ReQuest |
| 5 | R/W | Read/Write enable |
| 6 | Dot clock | |
| 7 | I/O 1 | |
| 8 | GAME | |
| 9 | EXROM | |
| 10 | I/O 2 | |
| 11 | ROML | |
| 12 | BA | |
| 13 | DMA | |
| 14 | D7 | |
| 15 | D6 | |
| 16 | D5 | |
| 17 | D4 | 8 Data lines |
| 18 | D3 | |
| 19 | D2 | |
| 20 | D1 | |
| 21 | D0 | |
| 22 | GND | |

**Bottom**

| Pin | Type | Comments |
|---|---|---|
| A | GND | |
| B | ROMH | |
| C | RESET | Hardware RESET line |
| D | NMI | Non-Maskable Interrupt |
| E | S 02 | |
| F | A15 | |
| H | A14 | |
| J | A13 | |
| K | A12 | |
| L | A11 | |
| M | A10 | |
| N | A9 | |
| P | A8 | 16 Address lines |
| R | A7 | |
| S | A6 | |
| T | A5 | |
| U | A4 | |
| V | A3 | |
| W | A2 | |
| X | A1 | |
| Y | A0 | |
| Z | GND | |

## The Kernel In Action

The Kernel is a library of I/O subroutines, which are called from BASIC or by user-written machine code. The Kernel ROM lies at $E000 to $FFFF, but the actual routines are called from a 'jump table' — that is, a table of pointers to the locations of the routines, held at the top of memory (see page 1138). The advantage of calling the routines via the jump table is that the code can be more easily used with other Commodore machines, because the addresses of the jump table do not vary, although the contents may, of course, change.

To use the kernel programs you will need reference material, since each routine needs to be carefully set up before it is called. Commodore's own Programmer's Reference Guide is probably the best source; kernel routines are documented on pages 268 to 306. For example, to use the kernel LOAD routine to perform a relocating LOAD the following code fragment is typical. First place the ASC (Commodore's version of ASCII) codes of the file name in some consecutive addresses in memory, then use the following piece of code:

```
LDA  #$01    Logical file number (lfn)
LDX  #$08    Device number (disk)
LDY  #$00    Secondary address (#$00
             gives relocating load)
JSR  $FFBA   Kernel routine SLFS (set lfn &
             secondary address)
LDA  #$0A    File name length (say decimal 10)
LDX  PL0     Lo-byte pointer to start address of
             file name
LDY  PHI     Hi-byte pointer to start address of
             file name
JSR  $FFBD   Kernel routine SETNAM (set file
             name)
LDA  #$00    Load = #$00 / Verify = #$01
LDX  DL0     Destination start address lo-byte
LDY  DHI     Destination start address hi-byte
JSR  $FFD5   Kernel routine LOAD
```

# Using RS232 On The Commodore 64

Provided a methodical approach is adopted there are no serious problems in using the OS routines to drive RS232, either from BASIC or machine code. There are several points to consider in preparing to use RS232 through the user port. Apart from the baud rate, which we'll come back to, the only other problems you are likely to encounter involve the following points:

- The Commodore 64 functions at 0v to 5v levels, whereas the RS232 standard requires -12v to 12v levels. Hence, unless communicating with another Commodore 64, it will be necessary to build or acquire a 'level conversion' facility. Commodore provide a RS232 cartridge to fulfil this function.
- Commodore ASC codes differ from the standard ASCII codes, so it will be necessary to set up two conversion arrays — one for transmitting and one for receiving.
- Whenever an RS232 channel is OPENed, the OS performs an automatic register clearance (CLR). The values of all variables previously used by a BASIC program are immediately lost; GOSUB commands produce error messages when RETURN is reached. This is because the RS232 kernel routine allocates two 256-byte buffers at the top of memory. This, in itself, can cause a problem because if there is insufficient space at the top of memory for the buffers, your program will be corrupted and errors will not be reported.
- If a BASIC program is long and/or makes a lot of string assignments, then sooner or later a garbage collection will occur. Incoming data may then be lost. This won't often happen, however, when the program is of modest length and doesn't make lots of string assignments.

To OPEN an RS232 channel from BASIC the following format is used:

```
OPEN2,2,3,CHR$(CTRL)+CHR$(COMM)
```

CTRL and COMM are the control and command bytes, which contain the information required to set up the channel. Note that CTRL and COMM must be two literal bytes (or the PEEKs of two previously filled locations), and not two variables! Each bit in the CTRL and COMM bytes has a function as shown in the tables.

For example, to OPEN an RS232 channel with one stop bit, a 7-bit word length, 300 baud (CTRL byte total =38), even parity, full duplex and 3-line flow control (COMM byte total = 96) we use the command:

```
OPEN2,2,3,CHR$(38)+CHR$(96)
```

When deciding what baud rate to use, the following factors must be kept in mind. When sending information, the baud rate is not critical, because other devices are usually tolerant of speed variations. Characters have been successfully sent via RS232 from BASIC at up to 2,400 baud. Of course, while the bit rate per byte is 2,400, the gap between bytes is often much longer, so the effective transfer rate is much less.

The situation is quite different, however, when receiving data. In this case, even at 300 baud, a BASIC program will barely have time to snatch a byte from the input buffer and print it on the screen within a simple loop. To receive effectively you must use some 'flow control'. This means telling the other device to stop sending before the input buffer fills up. The general procedure using 3-line protocol is:

1) Read in a small number of bytes (the higher the baud rate the smaller this number will have to be) using GET#2,A$. Deal with them quickly or buffer them by storing in an array for processing later.
2) Stop other devices sending using the command: PRINT#2,CHR$(17).
3) Read more bytes until the real RS232 buffer is empty. Process all bytes read in at your leisure, while checking for keypresses or end of file (EOF) signals etc.
4) Allow the device to send again using PRINT#2, CHR$(19) and go back to step one.

Once an RS232 channel has been OPENed, bytes are sent or received in the usual way, using PRINT# or GET#. INPUT# should be avoided. The status byte, ST, should be checked, as for any program involving I/O — the zero error state is denoted by ST=0 or ST=8. Finally, CLOSE can be expected to produce another auto-CLR, as the buffers are de-allocated.

**CTRL Byte**

| Bit 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Function |
|---|---|---|---|---|---|---|---|---|
| — | — | — | X | 0 | 0 | 0 | 1 | 50 baud |
| — | — | — | X | 0 | 0 | 1 | 0 | 75 baud |
| — | — | — | X | 0 | 0 | 1 | 1 | 110 baud |
| — | — | — | X | 0 | 1 | 0 | 0 | 134.5 bauc |
| — | — | — | X | 0 | 1 | 0 | 1 | 150 baud |
| — | — | — | X | 0 | 1 | 1 | 0 | 300 baud |
| — | — | — | X | 0 | 1 | 1 | 1 | 1200 baud |
| — | — | — | X | 1 | 0 | 0 | 0 | 1800 baud |
| — | — | — | X | 1 | 0 | 1 | 0 | 2400 baud |
| — | 0 | 0 | X | — | — | — | — | 8-bit data |
| — | 0 | 1 | X | — | — | — | — | 7-bit data |
| — | 1 | 0 | X | — | — | — | — | 6-bit data |
| — | 1 | 1 | X | — | — | — | — | 5-bit data |
| 0 | — | — | X | — | — | — | — | 1 stop bit |
| 1 | — | — | X | — | — | — | — | 2 stop bits |

X = Don't care

**COMM Byte**

| Bit 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Function |
|---|---|---|---|---|---|---|---|---|
| — | — | — | — | X | X | X | 0 | 3-line protocol |
| — | — | — | — | X | X | X | 1 | X-line protocol |
| — | — | — | 0 | X | X | X | — | Full duplex |
| — | — | — | 1 | X | X | X | — | Half duplex |
| — | — | 0 | — | X | X | X | — | Parity disabled |
| 0 | 0 | 1 | — | X | X | X | — | Odd parity |
| 0 | 1 | 1 | — | X | X | X | — | Even parity |
| 1 | 0 | 1 | — | X | X | X | — | Mark send p-check disabled |
| 1 | 1 | 1 | — | X | X | X | — | Space send p-check disabled |

# DATABASE

Here, courtesy of Acorn Computers, we publish another instalment in a series listing the call addresses of the BBC Micro's BASIC ROM routines. Note that these addresses are for BASIC II only

| | |
|---|---|
| PAGE statement routine | 9283 |
| PI function routine | ABCB |
| PLOT statement routine | 93F1 |
| POINT function routine | AB41 |
| POS function routine | AB6D |
| PRINT statement routine | 8D9A |
| PRINT# statement routine | 8D2B |
| PROC statement routine | 9304 |
| PTR function routine | BF47 |
| PTR statement routine | BF30 |
| Pack FAC#1 to &46C onwards | A385 |
| Pack FAC#1 to &471 onwards | A37D |
| Pack FAC#1 to &476 onwards | A381 |
| Pack FAC#1 to (&4B) | A38D |
| Parameters; unstacking | 8CC1 |
| Print A in hexadecimal | B545 |
| Print a character | B50E |
| Print a string | BFCF |
| Print number in decimal | 9EDF |
| Print number in hexadecimal | 9E90 |
| Print the IAC | 991F |
| Pull a string from stack | BDCB |
| Pull the IAC from stack | BDEA |
| Push FAC#1 onto stack | BD51 |
| Push a string onto stack | BDB2 |
| Push anything onto stack | BD90 |
| Push the IAC onto stack | BD94 |
| RAD function routine | ABB1 |
| READ statement routine | BB1F |
| REM statement routine | 8B7D |
| RENUMBER command routine | 8FA3 |
| REPEAT statement routine | BBE4 |
| REPORT statement routine | BFE4 |
| RESTORE statement routine | BAE6 |
| RETURN statement routine | B8B6 |
| RIGHT$ function routine | AFEE |
| RND function routine | AF49 |
| ROM Entry point | 8000 |
| ROM header | 8005 |
| RUN statement routine | BD11 |
| Real addition | 9C8B |
| Real comparison | 9A50 |
| Real multiplication | 9D20 |
| Real subtraction | 9CE1 |
| Remove FAC#1 from stack | BD7E |
| Remove a string from stack | BDDC |
| SAVE command routine | BEF3 |
| SGN function routine | AB88 |
| SIN function routine | A998 |
| SOUND statement routine | B44C |
| SPC routine | 8E58 |
| SQR function routine | A7B4 |
| STOP statement routine | 8AD0 |
| STR$ function routine | B094 |
| STRING$ function routine | B0C2 |
| Search for FN/PROC in catalogue | 945B |
| Search for a program line | B99A |
| Search for line in program | 9970 |
| Search for name in catalogue | 9469 |
| Search for next DATA item | BB50 |
| Search program for a PROC/FN | B112 |
| Series evaluator | A897 |
| Set (&2A)=(&2A)*(&3F) | 9236 |
| Set (&4B)=&46C | A7F5 |
| Set (&4B)=&471 | A7E9 |
| Set (&4B)=&476 | A7ED |
| Set FAC#1=(&4B)+FAC#1 | A500 |
| Set FAC#1=(&4B)-FAC#1 | A4FD |
| Set FAC#1=(&4B)/FAC#1 | A6AD |
| Set FAC#1=0 | A686 |
| Set FAC#1=1 | A699 |
| Set FAC#1=1/FAC#1 | A6A5 |
| Set FAC#1=A | A2ED |
| Set FAC#1=FAC#1*(&4B) | A606 |
| Set FAC#1=FAC#1*(&4B);chk ov/flow | A656 |
| Set FAC#1=FAC#1*10 | A1F4 |
| Set FAC#1=FAC#1*FAC#2 | A613 |
| Set FAC#1=FAC#1+FAC#2 | A50B |
| Set FAC#1=FAC#1-(&4B) | A4D0 |
| Set FAC#1=FAC#1-INT(FAC#1) | A486 |
| Set FAC#1=FAC#1/(&4B) | A6E7 |
| Set FAC#1=FAC#1/FAC#2 | A6F1 |
| Set FAC#1=FAC#1^A | AB12 |
| Set FAC#1=FAC#2 | A4DC |
| Set FAC#1=IAC | A2BE |
| Set FAC#2=0 | A453 |
| Set FAC#2=FAC#1 | A21E |
| Set FAC#2=FAC#1*2 | A23F |
| Set IAC=INT(FAC#1) | A3E4 |
| Set MAN#1=MAN#1*10 | A197 |
| Set MAN#1=MAN#1+A | A2A4 |
| Set MAN#1=MAN#1+MAN#2 | A178 |


© 1983 JULICH, J.—SHERIDAN COLLEGE